A Proof Assistant for PSF

S. Mauw & G.J. Veltink

Programming Research Group
University of Amsterdam
Kruislaan 403
1098 SJ Amsterdam
The Netherlands

Abstract
A description of a tool to support computer-aided construction of proofs for parallel systems is given. In contrast to the conventional approach based on state space exploration, we use an axiomatic approach. The axioms we use for the construction of proofs, are based on ACP. Besides these standard axioms we also consider tactics for shortening proofs. We use PSF (Process Specification Formalism), an extension of ACP with abstract data types, to describe the processes subject to the verification.

1. INTRODUCTION

One of the advantages of the use of formal techniques for the specification of parallel systems is that it enables formal verification of the correctness of such a specification. There are several approaches towards verification. One can verify certain properties of a specification, such as deadlock-freedom, fairness or starvation-freedom. A more general approach is to verify the truth of logic propositions about the execution traces of a specified system, see for example [HM85,TT91]. We will focus on a third approach, namely verifications of equality of two specifications, as developed in [Bae90]. Equality in this context can be interpreted in many ways, depending on the desired semantics.

A common way of proving that two processes are equal is by interpreting (or defining) the processes in some model, typically a graph model, followed by testing whether the interpretations are equal with respect to some congruence relation, such as observational equivalence or weak bisimulation. For finite process graphs several more or less efficient algorithms have been developed for determining these congruences [Fer91,GV90]. All of these algorithms suffer from the so-called state explosion problem. This problem comes from the fact that the number of states in a complex system is proportional to the product of the number of states of its parallel components.

An alternative is the algebraic or axiomatic approach, where a process expression is manipulated and proven equal to another process expression at a syntactic level, using an effectively given set of axioms. The advantage of this method over exploring the state space is that one can reason about the components or subsystems at a higher level of abstraction. Subsystems can be replaced by simpler ones and this way of pruning in the state graph results in simpler proofs. Another advantage is that an algebraic approach gives more insight into the reasons why a proof works or fails. This might give clues as to how faulty specifications can be repaired, and how correct specifications can be optimized. The axiomatic approach is also used in the PAM project [Lin91].

The restriction to finite state machines, or the class of regular processes, that is implied by the state exploration methods does not apply to the axiomatic approach. This way more complex processes, such as an unbounded queue, can be considered. The main

Note: This work was partially supported by ESPRIT Project no. 3006, CONCUR.

drawback of an axiomatic approach is that an equality guaranteed by some state space exploration algorithm need not be effectively constructable in the axiomatic system.

Computer tools supporting the axiomatic approach can be divided into two classes: theorem provers and proof assistants. The distinction is based on the level of mechanization of the process of proving. A simple proof assistant will have the form of an "electronic notebook" with accompanying software, which helps in rewriting the formulas that constitute the proof and will depend heavily on the interaction between people and machines. A more sophisticated theorem prover would make use of a number of heuristics to decide automatically what axioms to apply in what order.

ACP (Algebra of Communicating Processes) [BW90] is a process theory, which has been developed from an axiomatic viewpoint. Verifications of systems specified in ACP can be found in [Bae90] for example. The process specification language PSF [MV90] is an extension of ACP with abstract data types. It has a computer readable format and several computer tools have been developed to support specification in PSF, such as a syntax checker and a simulator.

In this paper we will describe how a proof assistant for PSF can be designed and the status of current preliminary investigation. This tool will be an aid in editing process expressions, selecting axioms that are applicable and applying these axioms. Preferably, sequences of applications of axioms which are commonly used in proofs must be offered using some shorthand. We will call these sequences: *tactics*.

This article is organised in the following way. We start off with a description of the toolkit for PSF which is under development followed with a short introduction to ACP. After that we will explain what we consider a proof within the proof assistant, give the axioms that are used to construct proofs and discuss the tactics that have been implemented. We conclude with a description of the implementation and an example to demonstrate the current status of the proof assistant.

2. THE PSF PROJECT

The PAT (Process Algebra Tools) project aims at constructing an environment of computer tools for studying concurrent systems, especially in the setting of the formal concurrency theory ACP (Algebra of Communicating Processes) [BW90]. Several tools had been written before the PAT project started, but because of the lack of a unified guiding framework there were many inconsistencies between the tools. The first step towards the construction of an integrated system in PAT has been the development of a language for specifying ACP-like processes in general. The resulting language PSF (Process Specification Language) is a formal specification language suitable for specifying concurrent systems. An introduction to the subject including examples is [MV89a], and the formal definition of PSF is given in [MV90].

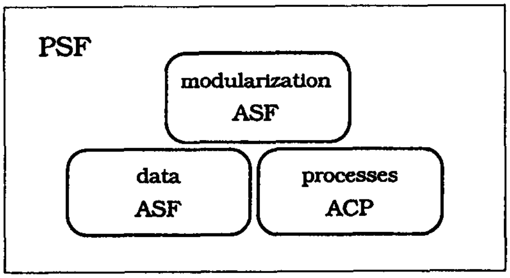

Figure 1.

PSF has been designed as the high-level specification language in the PAT project. It combines ACP with abstract data types. On the one hand PSF is based on ACP, that is for the part that is used to describe processes. The syntax of this part is kept as close as possible to the more informal syntax of ACP. On the other hand PSF is based on ASF

(Algebraic Specification Formalism) [BHK89]. This formalism is used to describe the data types with which processes can be parameterized. PSF also inherits its modularization concepts and its support of generics from ASF. Figure 1 gives a graphical representation of the constituting parts of PSF.

In contrast to PSF, a low-level language called TIL (Tool Interface Language) [MV89b] has been designed. TIL serves as a common kernel language for all the tools to be supported by the environment, including: a simulator; a proof assistant; a term rewriter and a bisimulation verifier. Although TIL was primarily intended as a dedicated interface language for the tools in the PAT project, it was designed so that it could be used on its own. From a semantical point of view TIL has the same expressive power as the combination of ACP and ASF. The main advantage of using TIL is that many parts of the toolkit can be reused. Because TIL is used mainly by tools, its readability for humans is of secondary importance.

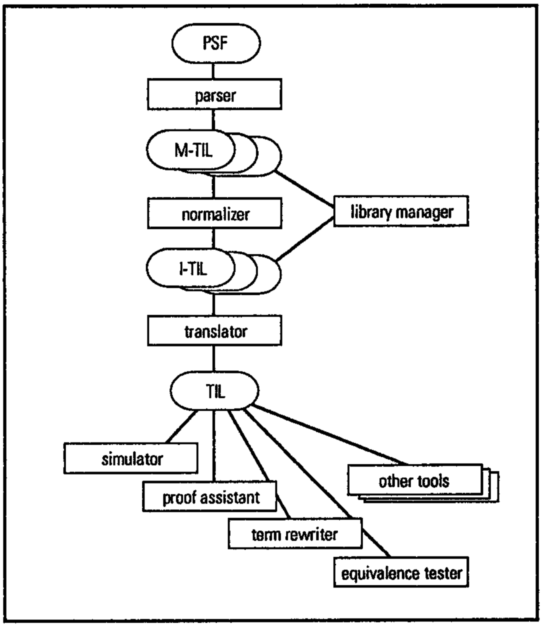

Figure 2.

At the centre of the toolkit, see figure 2, is the Tool Interface Language (TIL) through which all tools can communicate. From the picture we see that the PSF specification at the top is translated into TIL using two intermediate languages called M-TIL and I-TIL. In the course of this translation the library manager is used which supports and controls separate compilation of PSF modules.

Each PSF module is translated into exactly one M-TIL module. M-TIL is similar to TIL, but it still contains information about the modular structure of the specification. Because TIL supports no modular structure at all, a PSF specification has to flattened. This is done in the normalization phase in which the I-TIL language is used. The complete description of the translation from PSF to TIL can be found in [Vel90].

At the bottom of the picture we see the several tools. At the moment a simulator, a term rewriter and a proof assistant have been implemented. We are currently working on interfacing the toolkit with an existing tool for equivalence testing. Future plans for other tools include, for example, a compiler that compiles a PSF specification into a traditional programming language.

This approach of implementing an environment by using clearly defined intermediate languages, serves several purposes. The main reason is that it results in a layered design, in which humans can inspect specifications on a high level through PSF and in which the tools have access to the specifications through a low level representation tailored to their needs. This means in particular that the process of parsing and type checking of PSF is of no concern to the tools which will use a very simple parser to read the intermediate language. The second reason for using TIL and its derivatives is that having a definition of an intermediate language, construction of software can be started in parallel and so for example the construction of the simulator had not to wait for the parser and normalizer to be completed. The final reason for using TIL is that for new versions of PSF, or formalisms with comparable functionality, the toolkit can be easily adapted. Writing a new front-end for the specific language will be sufficient. In this way reusability of large parts of software, present in the tools, is guaranteed.

3. ACP

In this section we will give a brief introduction to ACP. This introduction is by no means intended to be complete. For more specific information on ACP we refer to [BW90]. The notation used here will differ slightly from the one used in the aforementioned book, because we have to deal with a computer readable syntax.

ACP starts from a set of objects, called atomic actions, atoms or steps. Atomic actions are the basic and indivisible elements of ACP. The (finite) set of constants is called A. On this set A there is defined a fixed partial function $\gamma : A \times A \rightarrow A$. Moreover we have two special constants:

- delta or deadlock. (delta $\notin$ A)
  deadlock is the acknowledgement that there is no possibility to proceed.
- skip or silent action. (skip $\notin$ A)
  skip represents the process terminating after some time, without performing any observable action.

Processes are constructed by combining constants and processes by operators. In the following introduction of the operators, *a,b,c* will stand for constants and *x,y,z* for processes.

- +, alternative composition or sum.
  $x + y$ is the process that first makes a choice between its summands x and y, and then proceeds with the execution of the chosen command. In the presence of an alternative, deadlock is never chosen.
- $\cdot$, sequential composition or product.
  $x \cdot y$ is the process that executes x first and continues with y after termination of x.
- $\|$, parallel composition or merge.
  $x \| y$ is the process that represents the simultaneous execution of x and y.
- $\lfloor\!\lfloor$, left merge.
  $x \lfloor\!\lfloor y$ is the process that represents the simultaneous execution of x and y in which the first action to be performed must come from x.
- $|$, communication merge.
  $x \mid y$ is the process that represents the simultaneous execution of x and y in which the first action to be performed must be a communication between an action from x and an action from y.
- encaps(H,x), encapsulation.
  encaps(H,x) is the process x without the possibility of performing from the set of actions H ($H \subseteq A$).
- hide(I,x), abstraction.
  hide(I,x) is the process x without the possibility of observing actions from the set of atomic actions I ($I \subseteq A$). This is achieved by renaming all atoms from I in x into *skip*.

4. PROOFS

In this section we will shortly discuss what we consider a proof in the setting of the proof assistant. A verification of a process expression consists of a stepwise transformation of this process expression into another one. This transformation can be seen as a proof that both expressions are equal in a certain process semantics. The semantics depends on the axioms used within the proof assistant. In the current implementation we use the weak bisimulation semantics [BW90].

In the next section we will give the axioms used by the proof assistant. If every step in a proof can be motivated by one of the given axioms, it can be considered correct. Using this technique, correctness of, say, a communication protocol is demonstrated by constructing a proof that the protocol specification and the service specification denote the same process. Computer support in construction of proofs can be applied in the editing of an expression, the selection of suitable axioms, the application of trivial transformation sequences, the report generation of the proof and to check manually constructed proofs.

5. AXIOMS

The axioms for ACP, describing the operators presented in section 3, can be transformed into a complete term rewriting system modulo commutativity and associativity [AB90]. The axioms given here should also be read as a TRS in which the expressions on the lefthand-side are rewritten into the expressions on the righthand-side. Some rules are added that are not present in the standard axiomatization, but they serve to optimize the rewriting within the proof assistant. See LMRG_SEQMRG for an example of an equality that, although it can be derived from the basic axioms, is a useful property to shorten proofs.

$x + y = y + x$	ALT_COMM
$x + (y + z) = (x + y) + z$	ALT_ASOC
$x + x = x$	ALT_IDENT
$x + \mathrm{delta} = x$	DLK_ALT

In the axioms we choose the left-associative form of an expression. See for example axiom ALT_ASOC. An exception to the this rule is SEQ_ASOC where the right-associative form is chosen because one is mostly interested in the first atom (head) of the sequential composition.

$(x + y) \cdot z = x \cdot z + y \cdot z$		SEQ_ALT
$(x \cdot y) \cdot z = x \cdot (y \cdot z)$		SEQ_ASOC
$\mathrm{delta} \cdot x = \mathrm{delta}$		DLK_SEQ
$x \parallel y = y \parallel x$		MRG_COMM
$x \parallel (y \parallel z) = (x \parallel y) \parallel z$		MRG_ASOC
$x \parallel y = (x \lfloor\!\lfloor y + y \lfloor\!\lfloor x) + x \mid y$		MRG_LMRG
$a \lfloor\!\lfloor x = a \cdot x$		LMRG_SEQ
$a \cdot x \lfloor\!\lfloor y = a \cdot (x \parallel y)$		LMRG_SEQMRG
$(x + y) \lfloor\!\lfloor z = x \lfloor\!\lfloor z + y \lfloor\!\lfloor z$		LMRG_ALT
$x \lfloor\!\lfloor \mathrm{delta} = x . \mathrm{delta}$		LMRG_DLKSEQ
$\mathrm{delta} \lfloor\!\lfloor x = \mathrm{delta}$		DLK_LMRG
$a \mid b = \gamma(a,b)$	if γ is defined	CMM_DEF
$a \mid b = \mathrm{delta}$	if γ is undefined	CMM_UNDEF
$x \mid \mathrm{delta} = \mathrm{delta}$		DLK_CMM
$x \mid y = y \mid x$		CMM_COMM
$(a \cdot x) \mid b = (a \mid b) \cdot x$		CMM_SEQ

(a · x) | (b · y) = (a | b) · (x ∥ y)		CMM_SEQMRG
(x + y) | z = x | z + y | z		CMM_ALT
encaps(H, a) = a	if a ∉ H	ENC_ATM
encaps(H, a) = delta	if a ∈ H	ENC_DLKATM
encaps(H, skip) = skip		ENC_SKP
encaps(H, x + y) = encaps(H, x) + encaps(H, y)		ENC_ALT
encaps(H, x · y) = encaps(H, x) · encaps(H, y)		ENC_SEQ
hide(I, a) = a	if a ∉ I	HID_ATM
hide(I, a) = skip	if a ∈ I	HID_DLKATM
hide(I, skip) = skip		HID_SKP
hide(I, x + y) = hide(I, x) + hide(I, y)		HID_ALT
hide(I, x · y) = hide(I, x) · hide(I, y)		HID_SEQ
x · skip = x		SKPACP_T1
skip · x + x = skip · x		SKPACP_T2
a · (skip · x + y) = a · (skip · x + y) + a · x		SKPACP_T3
x · (skip · y) = x · y		SKPACP_T1B

At the end of the table we have added some laws for *skip* which are known as Milner's τ-laws. Technically speaking the last axiom is not necessary, because it is a consequence of the first axiom. We have added it however because of the right-associative form used for the sequential composition.

6. TACTICS

Trying to prove facts by using only the axioms provided can be a tiresome job and therefore error-prone. As a typical example we found in one of our first experiments with the initial implementation of the proof assistant, that a simple proof that takes seven steps when done with pencil and paper takes more than sixty steps when applying only one axiom at a time. It goes without saying that a successful proof assistant should provide means to shorten such proofs. We have tried to cope with this problem by trying to mimic the reasoning used by human provers. In doing this, however, we remain exact all the time, that is we do not want to rely on *heuristics* of any kind. In this section we will discuss some of the tactics that we have found and that are implemented in the proof assistant. These tactics were developed by analyzing a number of manual ACP verifications from [Bae90].

In order to keep up with the state explosion problem it is crucial to be able to prune the state space as soon as possible. One of the strategies to follow is to try to produce and remove deadlocks as early as possible. By applying the axiom delta · x = delta, we can prevent unnecessary rewriting within x. Deadlocks are created within encapsulation expressions when atomic actions are prohibited to occur because they are blocked. An atomic action that is blocked could escape the blocking because it can engage in a communication and get renamed. However, atomic actions that are blocked by an encapsulation operator and are not able to communicate can be renamed into deadlocks safely. We can get rid of the processes following a delta by applying the axiom: delta · x = delta. This strategy is called *find deadlocks* in the proof assistant. A related strategy is *remove deadlocks*. This strategy removes, possibly multiple, deadlocks in one step by creating as much deadlocks as possible followed by applying: delta + x = x.

The next strategy is what we think one of the main strategies humans apply in constructing proofs in an ACP setting. In this strategy we try to separate a term X into a *head*, the first atom that is possible to occur, and the rest of the term, its *tail*. In analogy this strategy is called *head-tail*. In general the resulting term is not simply a head followed by a tail but it is of the form: $h_1 \cdot t_1 + h_2 \cdot t_2 + \ldots + h_n \cdot t_n$. In trying to create the head-tail expression each process variable that is encountered, is expanded. This means that the lefthand-side of a process definition is replaced by the appropriate righthand-side.

In fact this head-tail strategy is a combination of a number of axioms which relate closely to the expansion theorem from [BW90]. For brevity we state this axiom in the case we have a merge with two components X and Y, defined by

$$\begin{aligned}
&\mathrm{X} = \mathrm{a}_1 \cdot \mathrm{X}_1 + \ldots + \mathrm{a}_n \cdot \mathrm{X}_n\\
&\mathrm{Y} = \mathrm{b}_1 \cdot \mathrm{Y}_1 + \ldots + \mathrm{b}_n \cdot \mathrm{Y}_m, \text{ then}\\
&\mathrm{X} \parallel \mathrm{Y} = \mathrm{a}_1 \cdot (\mathrm{X}_1 \parallel \mathrm{Y}) + \ldots + \mathrm{a}_n \cdot (\mathrm{X}_n \parallel \mathrm{Y}) +\\
&\qquad \mathrm{b}_1 \cdot (\mathrm{X} \parallel \mathrm{Y}_1) + \ldots + \mathrm{b}_m \cdot (\mathrm{X} \parallel \mathrm{Y}_m) +\\
&\qquad (\mathrm{a}_1 | \mathrm{b}_1) \cdot (\mathrm{X}_1 \parallel \mathrm{Y}_1) + (\mathrm{a}_1 | \mathrm{b}_2) \cdot (\mathrm{X}_1 \parallel \mathrm{Y}_2) + \ldots + (\mathrm{a}_n | \mathrm{b}_m) \cdot (\mathrm{X}_n \parallel \mathrm{Y}_m)
\end{aligned}$$

In case the merge is surrounded by an encapsulation operator, we have the following:

$$\mathrm{encaps}(\mathrm{H}, \mathrm{X} \parallel \mathrm{Y}) = \sum_{\{i | a_i \notin H\}} \mathrm{a}_i \cdot (\mathrm{X}_i \parallel \mathrm{Y}) + \sum_{\{j | b_j \notin H\}} \mathrm{b}_j \cdot (\mathrm{X} \parallel \mathrm{Y}_j) + \sum_{\{i,j | a_i | b_j \notin H\}} (\mathrm{a}_i | \mathrm{b}_j) \cdot (\mathrm{X}_i \parallel \mathrm{Y}_j)$$

Also other combinations of operators are supported.

As a corollary we have the *recursive head-tail* strategy. Here we have to be careful not just to try to apply head-tail on all sub-expressions otherwise we would be able to wind up in an endless recursion when we consider the following process e.g.: $X = a \cdot X$. Without explicitly stating these, a number of rules are built in that determine when to stop recursion.

Finally there are three strategies implemented that relate to the so-called conditional axioms [BBK87]. These axioms are very useful in breaking a complex specification down into subsystems. They support a modular approach towards verification.

Before giving the axioms we first have to introduce the notion of the alphabet $\alpha(x)$ of a process x. This is the collection of all atomic actions that process x can perform (see [BBK87] for a definition).

Since this notion is not decidable (see [BBK87]), the alphabet of process x will be approximated by the collection of all actions used in the specification of x or one of its sub-processes. This inexactness does not influence the validity of the axioms.

Another notation which will be used is the communication set $S \mid T$ of two sets of atoms, S and T. This is defined by

$$\mathrm{S} \mid \mathrm{T} = \{ \mathrm{a|b} \mid \mathrm{a} \in \mathrm{S}, \mathrm{b} \in \mathrm{T}\}.$$

The first conditional axiom deals with pushing encapsulations through a merge.

$$\mathrm{encaps}(\mathrm{H}, \mathrm{X} \parallel \mathrm{Y}) = \mathrm{encaps}(\mathrm{H}, \mathrm{X} \parallel \mathrm{encaps}(\mathrm{H}', \mathrm{Y})),$$

$$\text{where } \mathrm{H}' = \mathrm{H} - \{ \mathrm{a} \in \alpha(\mathrm{Y}) \mid ((\{\mathrm{a}\} | \alpha(\mathrm{X})) \cap \mathrm{H}^{\mathrm{c}}) \neq \varnothing\} - \{\mathrm{a} \mid \mathrm{a} \notin \alpha(\mathrm{Y})\}.$$

The set H' is derived from H by first deleting all elements which can take part in a communication of which the resulting action is not encapsulated. Secondly we delete the actions from H that are superfluous because they do not occur in Y.

The second conditional axiom deals with pushing hiding through a merge.

$$\mathrm{hide}(\mathrm{I}, \mathrm{X} \parallel \mathrm{Y}) = \mathrm{hide}(\mathrm{I}, \mathrm{X} \parallel \mathrm{hide}(\mathrm{I}', \mathrm{Y})),$$

$$\text{where } \mathrm{I}' = \mathrm{I} - \{ \mathrm{a} \in \alpha(\mathrm{Y}) \mid (\{\mathrm{a}\} | \alpha(\mathrm{X})) \neq \varnothing\} - \{\mathrm{a} \mid \mathrm{a} \notin \alpha(\mathrm{Y})\}.$$

The third axiom is a combination of the first two.

$$\mathrm{hide}(\mathrm{I}, \mathrm{encaps}(\mathrm{H}, \mathrm{X} \parallel \mathrm{Y})) = \mathrm{hide}(\mathrm{I}, \mathrm{encaps}(\mathrm{H}, \mathrm{X} \parallel \mathrm{hide}(\mathrm{I}', \mathrm{Y}))),$$

$$\text{where } \mathrm{I}' = \mathrm{I} - \mathrm{H} - \{ \mathrm{a} \in \alpha(\mathrm{Y}) \mid (\{\mathrm{a}\} | \alpha(\mathrm{X})) \neq \varnothing\} - \{\mathrm{a} \mid \mathrm{a} \notin \alpha(\mathrm{Y})\}.$$

These three axioms are easily proved correct for closed process expressions, using the conditional axioms from [BBK87].

The following example will clarify the use of these axioms. We consider an array of n components, serially connected to each other. Without giving a description of the behaviour of the components, we assume that each component has k states. Thus the parallel composition (before encapsulation and abstraction) has k^n states. Now assume that after encapsulation and abstraction of the complete parallel composition a fairly simple process with, say $n{\cdot}k$ states results, then we would have needed to visit these $n{\cdot}k$ states in order to reduce the system to the smaller size.

Now by applying the conditional axioms described above, we can focus on the subsystem consisting of the first two components, which has k^2 states, and reduce it to a system with $2{\cdot}k$ states. The following step would be to focus on the subsystem obtained by combining this newly derived component and the third component, and so on. The result of this operation is that by restricting oneself to only sub-systems, the number of states visited decreases significantly. In this example the order of states visited would be $n^2{\cdot}k^2$ instead of k^n.

7. THE IMPLEMENTATION OF THE PROOF ASSISTANT

As mentioned earlier the proof assistant is part of the PAT project. It has been developed on SUN workstations and is written in the programming language C [KR78] using the X-Windows system. The proof assistant is an interactive tool and currently we restrict ourselves to PSF specifications in which atomic actions can be parameterized with elements from finite data sets only.

The proof assistant uses TIL as input language so one has to translate a PSF specification into TIL using the PSF-compiler. After reading the TIL specification, the proof assistant interacts with the user through five windows which are described below.

PSF Window. This window displays a PSF version of the specification which is constructed by translating the input TIL back into PSF. The text can be scrolled using the mouse. *Verify Window.* This window shows all the steps of the proof constructed so far. In this window the user selects the subterms that are to be manipulated. *Operation Window.* This window contains several buttons among which the buttons to activate the tactics. The buttons are only active when there is a term selected in the Verify Window. *Rewrite Window.* Whenever a subterm is selected in the Verify Window, this window pops up and shows the rewrite actions possible according to the axioms. The desired action can be selected using the mouse. *Special Window.* This window contains some buttons for actions to: undo the last step, reset the complete verification, choose another term from the specification to rewrite, generate troff output of the proof from the Verify Window, quit the proof assistant.

8. VERIFICATION OF TWO ONE-BIT BUFFERS, AN EXAMPLE

Although an explanation of an interactive tool by means of a written text is less adequate than active hands-on experience, we will try to demonstrate the working of the proof assistant with an example. We will use the tools to construct a proof that a system of two one-bit buffers shows the same behaviour as one two-bit buffer. Facilities for handling data will not be used, since the current status of the tools will not allow this.

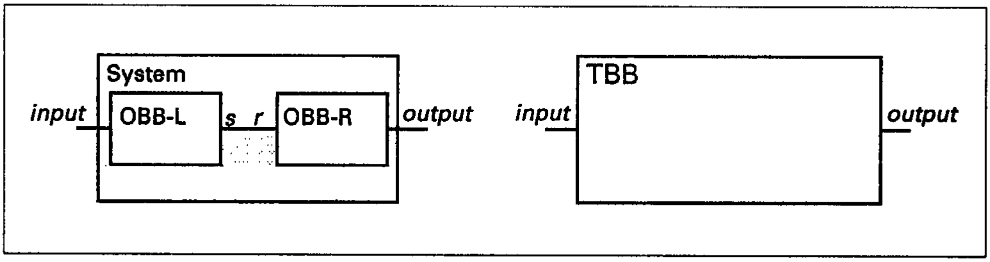

Figure 3.

THE SPECIFICATION

The system that we will consider consists of a parallel composition of two one-bit buffers, which are connected via an internal port. The left buffer (*OBB-L*) gets data from the environment via an *input*-port and sends it to the internal channel, while the right buffer reads data from this internal channel and hands it over to the environment via the *output*-port. The situation is as depicted in figure 3.

The shaded area is to indicate that we want to abstract from all actions at the internal port. In PSF the specification of this system looks as follows. First we define the behaviour of the two-bit buffer, which will be our target specification. The atomic actions *input* and *output* are defined, the process *TBB* which represents the two-bit buffer, and two auxiliary processes *TBB'* and *TBB''*.

The behaviour of the buffer is straightforward. It starts with an input action and comes in the state *TBB'* which indicates that there is one item in the buffer. *TBB'* can either do another input action and continue in state *TBB''* with two buffered items, or it can do an output action and restart with an empty buffer. Process *TBB''* can only do an output action and continue with *TBB'*.

For the system of two one-bit buffers, we define the processes *OBB-L* and *OBB-R*. Communication via the internal channel takes place by means of the *r* and *s* action. If both an *r* and an *s* action occur, this will result in a *c* action, which indicates successful communication.

The behaviour of the two one-bit buffers is defined straightforward. Now the System is defined as the parallel composition of these two buffers, while encapsulating unsuccessful communications (from the set *H*) and abstracting from communications via the internal channel (see the set *I*).

```
process module TBB
begin

  exports
    begin
      atoms
        input, output
      processes
        TBB, TBB', TBB''
    end

  definitions

    TBB   = input . TBB'
    TBB'  = input . TBB'' + output . TBB
    TBB'' = output . TBB'

end TBB
```

```
process module Buffers
begin

  imports
    TBB

  atoms
    s, r, c

  processes
    System, OBB-L, OBB-R

  sets of atoms
    H = { r, s }
    I = { c }

  communications
    s | r = c

  definitions

    OBB-L = input . s . OBB-L
    OBB-R = r . output . OBB-R
    System = hide(I,
                 encaps(H, OBB-L || OBB-R ))

end Buffers
```

VERIFICATION

The aim is to verify that the processes *TBB* and *System* define the same process, by which one may conclude for example that a composition of two one-bit buffers can be used as an implementation for a two-bit buffer. Figure 4 contains the output of the tool.

After starting the tool one can select the process to be manipulated. This will be the System process, for which the definition is displayed. After clicking on the *hide* operator to select the entire expression, applying the head-tail operation yields expression A1. The *skip* action comes out when doing this again (A2). After selecting the first dot, an axiom can be chosen which removes internal *skip* actions in this context (A3). The last step is to attach a new name, *S'* for example, to the expression after the input action (A4).

Next we focus on the newly defined process *S'*. After three steps we have been able to prove it equal to an expression which contains a new process name *S''* and the already defined process *System*. Note that the order of the left and the right buffer in the definition of System is opposite to the order in B2. However these two subexpressions are recognized by the proof assistant as being equal.

The third step is to repeat the process for the new process name *S''*. This yields an expression in which the definition for *S'* is recognized automatically (C3).

```
System = hide(I, encaps(H, OBB-L || OBB-R ) )
( A1 )       = input . hide(I, encaps(H, s . OBB-L || OBB-R ) )
( A2 )       = input . skip . hide(I, encaps(H, OBB-L || output . OBB-R ) )
( A3 )       = input . hide(I, encaps(H, OBB-L || output . OBB-R ) )
( A4 )       = input . S'

S' = hide(I, encaps(H, OBB-L || output . OBB-R ) )
( B1 )       = input . hide(I, encaps(H, s . OBB-L || output . OBB-R ) ) +
               output . hide(I, encaps(H, OBB-R || OBB-L ) )
( B2 )       = input . S'' + output . hide(I, encaps(H, OBB-R || OBB-L ) )
( B3 )       = input . S'' + output . System

S'' = hide(I, encaps(H, s . OBB-L || output . OBB-R ) )
( C1 )       = output . hide(I, encaps(H, OBB-R || s . OBB-L ) )
( C2 )       = output . skip . hide(I, encaps(H, output . OBB-R || OBB-L ) )
( C3 )       = output . hide(I, encaps(H, output . OBB-R || OBB-L ) )
( C4 )       = output . S'
```

Figure 4.

The result of this manipulation is that we have given a derivation that the process *System* is the solution of the following set of equations.

```
System       = input . S'
S'           = input . S'' + output . System
S''          = output . S'
```

Figure 5.

Now using the Recursive Specification Principle (see [BW90]) we can conclude that *System* and *TBB* in fact define the same process. This last step of reasoning has not yet been implemented in the proof assistant.

9. CONCLUSIONS

In this article we have given the description of a system that can be used to assist in the process of proving properties of process specifications. We think of the proof assistant as it is now, more as a somewhat smart electronic notebook than a full-fledged proof constructing system. It never has been our aim to be able to generate proofs automatically.

Even so we think there are still a large number of subjects on which the proof assistant can be improved. In the current version the axioms are 'hard-wired' into the code. The system would be more flexible if the user is allowed to enter a set of axioms of his own. In this way the user would also be able to select a different process semantics than the weak bisimulation that we have implemented. To be able to achieve this, a language for representing axioms has to be developed. Moreover one can think of an extension of the PSF language that allows to express proofs, which can be checked automatically afterwards.

We think that the conventional method of state space exploration and axiomatic approach should go hand in hand. In this way the axiomatic approach can be used to cut down the state space into several components which can then be checked by state space exploration. Although the implementation of the proof assistant has not been finished yet, we are encouraged by the fact that the tool is already used by people that are not involved in the PAT project. Experiences show that even for the relatively small examples, which it has been applied to, the tool is an important aid in constructing and analyzing specifications. Several suggestions for other tactics for special classes of specification domains are under consideration.

The authors would like to thank Bob Diertens for his practical work on the proof assistant and Ben Thompson for proofreading this paper and suggesting several improvements.

10. REFERENCES

[AB90] G.J. Akkerman & J.C.M. Baeten, Term rewriting analysis in process algebra, Report P9006, Programming Research Group, University of Amsterdam, 1990.

[Bae90] J.C.M. Baeten (ed.), Applications of Process Algebra, Cambridge Tracts in Theoretical Computer Science 17, Cambridge University Press, 1990.

[BBK87] J.C.M. Baeten, J.A. Bergstra & J.W. Klop, Conditional axioms and a/b-calculus in process algebra, in: Proceedings IFIP Conference on Formal Description of Programming Concepts III, Ebberup, (M. Wirsing, ed.) pp. 77-103, North-Holland, 1987.

[BHK89] J.A. Bergstra, J. Heering & P. Klint, The algebraic specification formalism ASF, in: Algebraic specification, J.A. Bergstra, J. Heering & P. Klint (eds.), pp. 1-66, ACM Press Frontier Series, Addison-Wesley 1989.

[BW90] J.C.M. Baeten & W.P. Weijland, *Process Algebra*, Cambridge Tracts in Theoretical Computer Science 18, Cambridge University Press, 1990.

[Fer91] J.C. Fernandez, *Aldébaran, A tool set for deciding bisimulation equivalences*, in: Proceedings CONCUR '91, Amsterdam, (J.C.M. Beaten & J.A. Bergstra, eds.), 1991. (to appear in LNCS series).

[GV90] J.F. Groote & F.W. Vaandrager, *An efficient algorithm for branching bisimulation and stuttering equivalence*, in: Proceedings 17th ICALP, Warwick, (M.S. Paterson, ed.) LNCS 443, pp. 626-638, Springer Verlag, 1990.

[HM85] M. Hennessy & R. Milner, *Algebraic Laws for Nondeterminism and Concurrency*, Journal of the Association for Computing Machinery, vol. 32, nr. 1, pp. 137-161, 1985.

[KR78] B.W. Kernighan & D.M. Ritchie, *The C programming language*, Prentice-Hall, 1978.

[Lin91] H. Lin, *PAM: A Process Algebra Manipulator*, this volume.

[MV89a] S. Mauw & G.J. Veltink, *An introduction to PSF_d*, in: Proc. International Joint Conference on Theory and Practice of Software Development, TAPSOFT '89, (J. Díaz, F. Orejas, eds.) LNCS 352, pp. 272-285, Springer Verlag, 1989.

[MV89b] S. Mauw & G.J. Veltink, *A Tool Interface Language for PSF*, Report P8912, Programming Research Group, University of Amsterdam, 1989.

[MV90] S. Mauw & G.J. Veltink, *A process specification formalism*, Fundamenta Informaticae XIII (1990), pp. 85-139, IOS Press, 1990.

[TT91] B.C. Thompson & J.V. Tucker, *Equational specification of Synchronous Concurrent Algorithms & Architectures*, University College of Swansea, Technical Report, 1991. (in preparation)

[Vel90] G.J. Veltink, *From PSF to TIL*, Report P9009, Programming Research Group, University of Amsterdam, 1990.